# Programming Constraint Inference Engines

Christian Schulte

Programming Systems Lab, DFKI, Stuhlsatzenhausweg 3, 66123 Saarbrücken, Germany
E-Mail: schulte@dfki.de, Web: www.ps.uni-sb.de/~schulte/

**Abstract.** Existing constraint programming systems offer a fixed set of inference engines implementing search strategies such as single, all, and best solution search. This is unfortunate, since new engines cannot be integrated by the user. The paper presents first-class computation spaces as abstractions with which the user can program inference engines at a high level. Using computation spaces, the paper covers several inference engines ranging from standard search strategies to techniques new to constraint programming, including limited discrepancy search, visual search, and saturation. Saturation is an inference method for tautology-checking used in industrial practice. Computation spaces have shown their practicability in the constraint programming system Oz.

## 1 Introduction

Existing constraint programming systems like CHIP [4], clp(FD) [2], ECLiPSe [1], and ILOG Solver [9] offer a fixed set of inference engines for search such as single, all, and best solution search. This is unfortunate, since new engines can only be implemented by the system's designer at a low level and cannot be integrated by the user.

The paper presents first-class computation spaces as abstractions with which the user can program inference engines at a high level. A computation space encapsulates a speculative computation involving constraints. Constraint inference engines are programmed using operations on computation spaces. They are made available as first-class citizens in the programming language to ease their manipulation and control.

We demonstrate that computation spaces cover a broad spectrum of inference engines ranging from standard search strategies to techniques new to constraint programming. The standard strategies discussed include single, all, and branch-and-bound best solution search. Their presentation introduces techniques for programming search engines using computation spaces. It is shown how these techniques carry over to engines like iterative deepening [10] and restart best solution search.

The paper studies limited discrepancy search, visual search, and saturation which are new to constraint programming. Limited discrepancy search (LDS) has been proposed by Harvey and Ginsberg [6] as a strategy to exploit heuristic information. A visual and interactive search engine that supports the development of constraint programs is the Oz Explorer [11]. It is an example where the user directly benefits from the expressiveness of computation spaces. The paper explains recomputation which is related to engines programmed from computation spaces. It allows to solve problems with a large number of variables and constraints that would need too much memory otherwise.

Appears in: Gert Smolka, editor, *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science, Schloß Hagenberg, Austria, October 1997. Springer-Verlag.

Saturation (also known as Stålmarck's method) is a method for tautology-checking used in industrial practice [5,14]. We show how a generic saturation engine can be built from first-class computation spaces. The engine is generic in that it is not restricted to a particular constraint system, whereas saturation has been originally conceived in the context of tautology-checking of Boolean formulae.

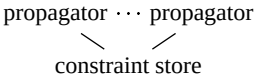
The constraint programming system Oz [8,13] implements computation spaces. While Oz offers computation spaces to the experienced user, it also provides a library of predefined search engines programmed with computation spaces that can be used without requiring detailed knowledge on computation spaces [7]. Search engines programmed from spaces are efficient. For example, solving scheduling problems using these engines is competitive with commercially available systems [16].

This work is based on a previous treatment of the so-called search combinator [12]. It spawns a local computation space and resolves remaining choices by returning them as procedures. First-class computation spaces subsume the search combinator and provide a more natural abstraction for programming inference engines. The paper contributes by introducing first-class computation spaces, but its main contribution is how to employ computation spaces for constraint inference engines that go beyond existing constraint programming systems.

The plan of the paper is as follows. The inference methods studied in the paper are shown in Section 2. Section 3 introduces first-class computation spaces. The rest of the paper is concerned with how the inference methods presented in Section 2 can be programmed with first-class computation spaces. Sections 4 to 10 introduce various search engines, whereas first-class computation spaces are applied to saturation in Section 11. Section 12 gives a brief conclusion.

## 2 Constraint Inference Methods

This section presents search and saturation as constraint inference methods using computation spaces as their central notion.




A *computation space* consists of propagators connected to a constraint store. The *constraint store* holds information about values of variables expressed by a conjunction of basic constraints. *Basic constraints* are logic formulae interpreted in a fixed first-order structure. In the following we restrict ourselves to finite domain constraints. A basic finite domain constraint has the form $x \in D$ where $D$ is a finite subset of the positive integers. Other relevant basic constraints are $x = y$ and $x = n$, where $n$ is a positive integer.

To keep operations on basic constraints efficient, more expressive constraints, called *nonbasic*, e.g., $x + y = z$, are not written to the constraint store. A nonbasic constraint is imposed by a propagator. A *propagator* is a concurrent computational agent that tries to amplify the store by *constraint propagation*: Suppose a constraint store hosting the constraint $C$ and a propagator imposing the constraint $P$. The propagator can *tell* a basic constraint $B$ to the store, if $C \wedge P$ entails $B$ and $B$ adds new and consistent information to $C$. Telling a basic constraint $B$ updates the store to host $C \wedge B$.

A propagator imposing $P$ disappears as soon as it detects that $P$ is entailed by the store's constraints. A propagator imposing $P$ becomes *failed* if it detects that $P$ is in-

consistent with the constraints hosted by the store. A space $S$ is *stable*, if no further constraint propagation in $S$ is possible. A stable space that contains a failed propagator is *failed*. A stable space not containing a propagator is *solved*.

*Distribution.* Typically, constraint propagation is not enough to solve a constraint problem: A space may become stable but neither solved nor failed. Hence, we need distribution. *Distributing* a space $S$ with respect to a constraint $D$ yields two spaces: One is obtained by adding $D$ to $S$ and the other is obtained by adding $\neg D$ to $S$. The constraint $D$ will be chosen such that adding of $D$ ($\neg D$) enables further constraint propagation.

*Search.* To solve a constraint problem with search, a space containing basic constraints and propagators of the problem is created. Then constraint propagation takes place until the space becomes stable. If the space is failed or solved, we are done. Otherwise, we select a constraint $D$ with which we distribute the space. A possible distribution strategy for finite domain constraint problems is: Select a variable $x$ which has more than one possible value left and an integer $n$ from these values and then distribute with $x = n$.

Iterating constraint propagation and distribution creates a tree of computation spaces ("search tree") where leaves are failed or solved spaces. In this setup, the search tree is defined entirely by the problem and the distribution strategy. An orthogonal issue is how the search tree is explored by a given search engine.

*Saturation.* Distributing a space $S$ yields two spaces $S_0$ and $S_1$ in which constraint propagation can tell new basic constraints to the stores of $S_0$ and $S_1$. The idea of saturation is to add basic constraints to $S$ by *combining* $S_0$ and $S_1$: add the basic constraints that are common to both $S_0$ and $S_1$ back to the original space $S$. This might enable further constraint propagation within $S$. Distribution of $S$ and combination back to $S$ is iterated until a fixed point is reached. In the saturation literature, distribution and combination together is referred to as so-called *dilemma rule*.

Saturation is commonly applied to propositional satisfiability problems involving Boolean variables (finite domain variables restricted to take either 0 (false) or 1 (true) as value). In this context, a possible distribution strategy is: select a variable $x$ from some fixed set of variables $X$ and proceed from a space $S$ by distribution to spaces $S_0$ by adding $x = 0$ and $S_1$ by adding $x = 1$. After constraint propagation has finished, $S_0$ and $S_1$ are combined as follows: For each variable $y \in X$ where both $S_0$ and $S_1$ contain the basic constraint $y = n$ ($n \in \{0,1\}$), the constraint $y = n$ is added back to $S$. Distribution and combination is iterated for all variables in $X$ until either $S$ becomes failed or an entire iteration over all variables in $X$ adds no new basic constraints to $S$. If one of $S_0$ or $S_1$ fails, saturation proceeds with the other space. Note that saturation, in contrast to search, is incomplete: after saturation finishes the space might not be solved.

Saturation as sketched above considers only a single variable at a time and thus is called 1-saturation. $n$-Saturation takes $n$ variables simultaneously into account: it recursively applies $(n-1)$-saturation to the spaces $S_0$ and $S_1$ obtained by distribution, where 0-saturation coincides with constraint propagation. In practice only 1-saturation and 2-saturation are used. Harrison reports in [5] that the unsatisfiability of many practical formulae can in fact be proved with 1-saturation.

## 3 First-Class Computation Spaces

Section 2 demonstrates that computation spaces are a powerful abstraction for constraint inference methods. To control and manipulate computation spaces as needed in inference engines, they should be available as first-class entities. The programming language Oz [8,13] provides computation spaces as first-class citizens. They can be passed as arguments of procedures, can be tested for equality and the like. They can be created, their status can be asked for, they can be copied, their constraints can be accessed, and additional constraints can be injected into them.

Besides constraint store and propagators, a computation space also hosts threads. Like propagators, threads are concurrent computational entities. A *thread* is a stack of statements. It runs by reducing its topmost statement, possibly replacing the reduced statement with new statements. Threads synchronize on their topmost statement: if the topmost statement cannot be reduced, the entire thread cannot be reduced; we say it *suspends*. Statements include procedure application, procedure definition, variable declaration, sequential composition of statements, tell statement, conditional statement, thread creation, propagator creation, and so-called choices. A *choice* is either unary (**`choice`** $S$ **`end`**) or binary (**`choice`** $S_1$ `[]` $S_2$ **`end`**) where $S$, $S_1$, and $S_2$ are statements called *alternatives*. A thread with a choice as topmost statement suspends.

Reduction of the statement `S={NewSpace P}` creates a new computation space, where `P` is a unary procedure and `S` yields a reference to the newly created space. The newly created space features a single fresh variable, the so-called *root variable*. In `S` a thread is created that contains as its single statement the application of `P` to the root variable. Typically, the procedure `P` represents the problem to be solved, and its single argument gives access to the solution of the problem (see the procedure `Money` in Section 5). Running the newly created thread typically creates variables, adds basic constraints to the store, spawns further propagators, and creates further threads.

A computation space `S` is *stable* if no thread and no propagator in `S` can reduce, and cannot become reducible by means of any other computation (more details on stability can be found in [12]). A computation space `S` is *failed* if it contains a failed propagator. When a computation space becomes stable and contains a thread with a unary choice as its topmost statement, the unary choice is replaced by its alternative. That is, a unary choice synchronizes on stability (this is used to program distribution strategies, see Section 6). A stable computation space not containing threads with unary choices but with binary choices as their first statements is called *distributable*. When a space becomes distributable one thread containing a binary choice as its topmost statement is selected. A stable space is *succeeded*, if it does not contain threads which suspend on choices.

A computation space `S` can be asked by `A={Ask S}` for its status. As soon as `S` becomes stable, the variable `A` gets bound. If `S` is failed then `A` is bound to `failed`. If `S` is distributable, `A` is bound to `alternatives`. Otherwise, `A` is bound to `succeeded`.

A distributable space `S` allows to commit to one alternative of the selected choice. By `{Commit S I}` the space `S` commits to the `I`-th alternative of the selected choice. That is, the choice is replaced by its `I`-th alternative. To explore both alternatives of a selected choice, stable computation spaces can be cloned. Reduction of `C={Clone S}` creates a new computation space `C` which is a copy of the stable space `S`.

The operation `{Inject S P}` takes a space `S` and a unary procedure `P` as arguments. Similar to space creation, it creates a new thread that contains as single statement the application of `P` to the root variable of `S`. For example, combination for saturation uses `Inject` to add constraints to an already existing space.

The constraints of a local computation space `S` can be accessed by `Merge`. Reduction of `{Merge S X}` binds `X` to the root variable of `S` and then discards `S`. The constraints that were referred to by the root variable of `S` can now be referred to by `X`.

The presentation here has been simplified in two aspects. Firstly, only binary choices are considered here. Oz in fact provides also for non-binary choices, this requires the operations `Ask` and `Commit` to be generalized in a straightforward manner. The second simplification concerns the setup of spaces in Oz. Regular computations in Oz are carried out in the so-called top-level computation space. Computations involving constraint propagation and distribution are speculative in the sense that failure is a regular event. These computations are encapsulated by first-class computation spaces. It is perfectly possible in Oz to create first-class spaces within first-class spaces (think of nested search engines) which leads to a tree of computation spaces (not to be confused with a search tree).

## 4 Depth-First Search

To familiarize the reader with programming inference engines using spaces, this section introduces simple depth-first search engines.

```
fun {DFE S}
   case {Ask S} of failed then nil
   [] succeeded then [S]
   [] alternatives then C={Clone S} in
      {Commit S 1}
      case {DFE S} of nil then {Commit C 2} {DFE C}
      [] [T] then [T]
      end
   end
end
```

**Fig. 1.** Depth-first single solution search.

The procedure `DFE` (see Figure 1) takes a space as argument and tries to solve it following a depth-first strategy. If no solution is found, but search terminates, the empty list `nil` is returned. Otherwise, the procedure returns the singleton list `[T]` where `T` is a succeeded computation space. Depending on the status of `S` (as determined by `Ask`), either the empty list `nil` or a singleton list containing `S` is returned. Otherwise, after distributing with the first clause (by `{Commit S 1}`) exploration is carried out recursively. If this does not yield a solution (i.e., `nil` is returned), a clone `C` of `S` is distributed with the second clause and `C` is solved recursively.

To turn the procedure DFE into a search engine that can be used easily, without any knowledge about spaces, we define the following procedure:

```
fun {SearchOne P}
   case {DFE {NewSpace P}} of nil then nil
   [] [S] then X in {Merge S X} [X]
   end
end
```

SearchOne takes a unary procedure as input, creates a new space in which the procedure P is run, and applies the procedure DFE to the newly created space. In case DFE returns a solved space, its root variable is returned in a list.

The search engine DFE can be adapted easily to accommodate for all solution search, where the entire search tree is explored and a list of all solved spaces is returned. It is sufficient to replace the shaded lines in Figure 1 with:

```
{Commit S 1} {Commit C 2} {Append {DFE S} {DFE C}}
```

Other depth-first search engines can be programmed following the structure of the program in Figure 1. For example, a search engine that puts a depth limit on the explored part of the search tree would just add support for maintaining the exploration depth (incrementing the depth on each recursive application of DFE). Increasing the depth limit until either a solution is found or the entire search tree is explored within the depth limit then yields iterative deepening [10].

## 5 An Example: Send Most Money

Let us consider a variation of a popular puzzle: Find distinct digits for the variables $S$, $E$, $N$, $D$, $M$, $O$, $T$, $Y$ such that $S \neq 0$, $M \neq 0$ (no leading zeros) and the equation $SEND + MOST = MONEY$ holds. The program for this puzzle is shown in Figure 2.

```
proc {Money Root}
   money(s:S e:E n:N d:D m:M o:O t:T y:Y) = Root
in
   Root ::: 0#9
   {FD.distinct Root} S\=:0 M\=:0
                S*1000 + E*100 + N*10 + D
              + M*1000 + O*100 + S*10 + T
   =: M*10000 + O*1000 + N*100 + E*10 + Y
   {FD.distribute ff Root}
end
```

**Fig. 2.** A program for the $SEND + MOST = MONEY$ puzzle.

The problem is defined as unary procedure, where its single argument Root is constrained to the solution of the problem. Here, the solution is a record that maps letters to variables for the digits. Execution of Root ::: 0#9 tells the basic constraints that

each field of `Root` is an integer between 0 and 9. The propagator `FD.distinct` enforces all fields of the record to be distinct, whereas the propagators `S\=:0` and `M\=:0` enforce the variables `S` and `M` to be distinct from zero. The gray-shaded lines display the propagator that enforces $SEND + MOST = MONEY$. The variables for the letters are distributed (by `FD.distribute`) according to a strategy as sketched in Section 2, where variable selection follows the first-fail heuristic: the variable with the smallest number of possible values is distributed first.

Applying the search engine (i.e., `SearchOne`) to the problem (i.e., `Money`) by `{SearchOne Money}` returns the following first solution:

```
[money(d:2 e:3 m:1 n:4 o:0 s:9 t:5 y:7)]
```

Search engines usually are not built from scratch. Oz comes with a library of predefined search engines [7] programmed from computation spaces. The interface between search engine and problem is well defined by the choices as created by the distribution strategy. The next section shows how to program distribution strategies with choices.

## 6 Programming Distribution Strategies

Figure 3 shows a distribution strategy similar to that mentioned in Section 2. The procedure `Distribute` takes a list of finite domain variables to be distributed. A unary choice is employed to synchronize the execution of the gray-shaded statement on stability of the executing space. Since a distribution strategy typically inspects the constraint store's current state, it is important that variable and value selection for choice creation takes place only after constraint propagation has finished.

```
proc {Distribute Xs}
   choice  case {SelectVar Xs} of nil then skip
           [] [X] then N={SelectVal X} in
              choice X=N [] X\=:N end {Distribute Xs}
           end
   end
end
```

**Fig. 3.** Programming a distribution strategy with choices.

After synchronizing on stability, `SelectVar` selects a variable `X` that has more than one possible value left, whereas `SelectVal` selects one possible value `N` for `X`. The binary choice states that either `X` is equal or not equal to `N`. The first-fail strategy as used in Section 5, for example, would implement `SelectVar` as to return the variable with the smallest number of possible values and `SelectVal` as to return its minimal value.

## 7 Best Solution Search

Best solution search determines a best solution with respect to a problem-dependent ordering among the solutions of a problem. It is important to not explore the entire

search tree of a problem but to use solutions as they are found to prune the rest of the search space as much as possible.

*An Example: Send Most Money.* Let us reconsider the example of Section 5. Suppose that we want to find the solution of the puzzle $SEND + MOST = MONEY$ where we can get the most money: *MONEY* should be as large as possible. For this, we define the binary procedure `More` that takes two root variables `O` and `N` and imposes the constraint that `N` is better than `O` ( `O.m` returns the variable at field `m` in record `O`):

```
proc {More O N}
   O.m*10000 + O.o*1000 + O.n*100 + O.e*10 + O.y <:
   N.m*10000 + N.o*1000 + N.n*100 + N.e*10 + N.y
end
```

We can search for the best solution by `{SearchBest Money More}` (`SearchBest` will be explained later), which returns a singleton list with the best solution as follows:

```
[money(d:2 e:7 m:1 n:8 o:0 s:9 t:4 y:6)]
```

Essential for best solution search is to inject to a space an additional constraint that the solution must be better than a previously found solution. The procedure

```
proc {Constrain S SolS O}
   {Inject S proc {$ NR}
                OR in {Merge {Clone SolS} OR} {O OR NR}
             end}
end
```

takes a space `S`, a solved space `SolS` (the so far best solution) and a binary procedure implementing the order (e.g., `More` in the above example). It injects into `S` the constraint that `S` must yield a better solution than `SolS` which is implemented by an order `O` on the constraints accessible from the root variables of the solution and the space `S` itself. Note that the solution's constraints are made accessible by merging a clone of `SolS` rather than merging `SolS` itself. This allows to use `SolS` again with `Constrain` and to possibly return it as best solution.

The procedure `BAB` (shown in Figure 4) implements branch-and-bound search. It takes the space `S` to be explored, the space `SolS` as the so far best solution, and the order `O` as arguments. It returns the space for the best solution or `nil` if no solution exists. Initially, `SolS` is `nil`. The procedure maintains the invariant that `S` can only lead to a solution that is better than `SolS`. In case that `S` is failed the so far best solution is returned. In case `S` is solved, it is returned as a new and better solution (which is guaranteed by the invariant). The central part is shaded gray: if following the first alternative returns a better solution (the invariant ensures that a different space is also a better one), the space for the second alternative is constrained to yield an even better solution than `SolS`. Note that here the unique identity of spaces is exploited and that `nil` is different from any space. The latter ensures that `Constrain` never gets applied to `nil`. A procedure `SearchBest` as used in the above example can be obtained easily. Like the search engines presented so far, it creates a space running the procedure to be solved, applies `BAB`, and possibly returns the best solution.

A different technique for best solution search is restart. After a solution is found, search restarts from the original problem together with the constraint to yield a better

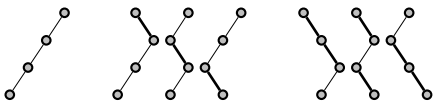
```
fun {BAB S SolS O}
   case {Ask S} of failed then SolS
   [] succeeded then S
   [] alternatives then
      C={Clone S} {Commit S 1} {Commit C 2}
      NewS={BAB S SolS O}
   in
      case NewS==SolS then skip else {Constrain C NewS O} end
      {BAB C NewS O}
   end
end
```

**Fig. 4.** Branch-and-bound best solution search engine.

solution. Suppose `S` is a space for the problem to be solved. A best solution can be computed by iterating application of a search engine to a clone of `S` and application of `Constrain` until no further solution is found. Then, the last solution found is best. Restart can be beneficial if it is easier to find a first solution rather than exploring large parts of the search tree to proceed from one solution to a better one by branch-and-bound. Any single solution search engine can be used together with the restart technique. For example, limited discrepancy search (see Section 8) together with restart can compute a solution that is an upper bound for a best solution. Then branch-and-bound search can be used to find a solution better than this upper bound.

## 8 Limited Discrepancy Search

Usually distribution strategies follow a heuristic that has been carefully designed to suggest most often "good" alternatives, where good alternatives are those leading to a solution. This is taken into account by limited discrepancy search (LDS), which has been introduced by Harvey and Ginsberg [6]. LDS has been successfully applied to scheduling problems [3]. Experimental results on frequency assignment problems [15] provide evidence for the potential of LDS in constraint programming.



Central to LDS is to *probe* the search tree with a fixed number of discrepancies. A *discrepancy* is a decision made during exploration of the search tree against the heuristic. In our setting a discrepancy thus amounts to committing to the second alternative of a choice first, rather than to its first alternative. *Probing* means to explore the search tree with a fixed number of discrepancies $d$. Probing for $d = 0, 1$, and 2 is sketched to the right, where discrepancies are shown as thick vertices (the illustration is adapted from [6]).

Since only little information is available at the root of the search tree, it is more likely for a heuristic to make a wrong suggestion there. Probing takes this into account by making discrepancies at the root of the search tree first. If this does not lead to a solution discrepancies are made further down in the tree (as sketched by the order in the illustration above).

```
fun {Probe S N}
   case {Ask S} of failed then nil
   [] succeeded then [S]
   [] alternatives then
      case N>0 then C={Clone S} in
         {Commit S 2}
         case {Probe S N-1} of nil then {Commit C 1} {Probe C N}
         [] [T] then [T]
         end
      else {Commit S 1} {Probe S 0}
      end
   end
end
```

**Fig. 5.** Probing for limited discrepancy search.

LDS now iterates probing with 0, 1, 2, ... discrepancies until a solution is found or a given limit for the discrepancies is reached.

Figure 5 shows `Probe` that implements probing. It takes a space `S` and the number of discrepancies `N` as input, and returns either the empty list in case no solution is found or a singleton list containing a solved space. The case where the space is failed or solved is as usual. If `S` is distributable and `N` is greater than zero, a discrepancy is made (by `{Commit S 2}`). If the recursive application of probing with one discrepancy less does not yield a solution, probing continues by making the discrepancy further down in the search tree. Otherwise (that is, `N` is zero) probing continues without any discrepancy.

It is interesting that the program shown in Figure 5 is close both in length and structure to the pseudo-code for probing in [6]. This demonstrates that spaces provide an adequate level of abstraction for search engines of this kind.

A complete implementation of LDS that takes a procedure `P` and a maximal limit `M` for the discrepancies as input can be obtained straightforwardly from `Probe`. Similar to `SearchOne` in Section 4 a space `S` running `P` is created. Then application of `Probe` to a clone of `S` and the number of allowed discrepancies is iterated until either a solution is found or the discrepancy limit `M` is reached.

## 9 The Oz Explorer: A Visual Search Engine

The Oz Explorer [11] is a visual search engine that supports the development of constraint programs. It uses the search tree as its central metaphor. The user can interactively explore the search tree which is visualized as it is explored. Visible nodes carry as information the corresponding computation space that can be accessed interactively by predefined or user-defined procedures. The Explorer also supports best solution search.

The Explorer is implemented using first-class computation spaces. Its main data structure is the search tree implemented as a tree of objects. Each node is an object that stores a computation space. The object's class depends on the status of its stored space (that is, whether the space is failed, solved, or distributable) and provides methods

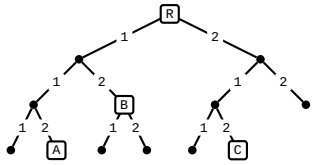
for exploration and visual appearance. Invoking an operation at the interface sends a message to the object and triggers execution of the corresponding method.

The Explorer demonstrates nicely that new and interesting search engines can be designed and programmed easily with computation spaces (as is reported in [11]). Interactive exploration presupposes that search is not limited to a depth-first strategy. User access to the computation state of a search tree's node requires that computation spaces are first-class. Hence, the user of the Explorer directly profits from the expressiveness of first-class computation spaces.

## 10 Recomputation: Trading Space for Time

When solving complex real-world problems, computation spaces might contain a large number of variables and constraints. Since `Clone` creates a copy of a space to be held in memory, search engines that create many clones might use too much memory. A drastic example is the Explorer: It needs to store all spaces in the already explored part of a search tree to provide user access to them. That leads to space requirements similar to breadth-first search. A solution to this problem is recomputation, where spaces are recomputed on demand rather than being cloned in advance.

```
fun {Recompute S Is}
   case Is of nil then
      {Clone S}
   [] I|Ir then
      C={Recompute S Ir}
   in
      {Commit C I} C
   end
end
```




```
A={Recompute R [2 1 1]}
B={Recompute R [2 1]}
C={Recompute R [2 1 2]}
```

**Fig. 6.** Recomputing spaces.

The procedure `Recompute` (see Figure 6) recomputes a space from a space `S` higher up in the search tree and a list of integers `Is` describing the path between the two spaces. The path is represented bottom-up, since it can be constructed easily that way by adding the alternative's number to the path's head during top-down exploration.

The most extreme version of recomputation is to always recompute from the search tree's root space. The procedure `DFE` as shown in Figure 1 can be extended by two additional arguments: `R` for the root space and `Is` for the path of the current space `S` to the root. Cloning is replaced by recomputation. Recursive applications of `DFE` additionally maintain the path to the root of the search tree. For example, the part of the search engine that explores the second alternative of a space looks as follows:

```
··· C={Recompute R Is} in {Commit C 2} {DFE C R 2|Is}
```

A more practical strategy for recomputation is the idea of a maximal recomputation distance *n*: Clone a space once in a while such that recomputation never must recompute more than *n* applications of `Commit`. For example, the Explorer uses this strategy.

## 11 A Generic Saturation Engine

This section shows how to build a generic saturation (for saturation see Section 2) engine from computation spaces. The engine is generic in the sense that it is not limited to Boolean problems.

*An Example.* As an example we apply saturation to check whether the Boolean formula $A \wedge (B \vee (A \wedge B))$ is unsatisfiable: Does $A \wedge (B \vee (A \wedge B)) \Leftrightarrow 0$ hold, where $\Leftrightarrow$ reads as equivalence and 0 as false. The first step is a translation into so-called triplets by introducing new Boolean variables $C$ and $D$: $A \wedge B \Leftrightarrow C$, $B \vee C \Leftrightarrow D$, and $A \wedge D \Leftrightarrow 0$. What is called triplets in the saturation literature we implement as Boolean propagators:

```
proc {P Xs} [A B C D]=Xs in
   Xs ::: 0#1
   {FD.conj A B C} {FD.disj B C D} {FD.conj A D 0}
   {SatDist Xs}
end
```

Here `Xs:::0#1` constrains the variables `A`, `B`, `C`, and `D` to take Boolean values and `FD.conj` (`FD.disj`) spawns a propagator for equivalence to a conjunction (disjunction). The details of `SatDist` are explained below. It is important to also take into account variables introduced for subformulas (`C` and `D` in our example) [5]. For example, distribution and combination on `C` can add information on both `A` and `B`.

During saturation the following happens. First, constraint propagation takes place but cannot add new constraints. Suppose that `SatDist` distributes first on `A`. Constraint propagation in the space for `A=0` adds `C=0` (by `{FD.conj A B C}`), in the space for `A=1` it first adds `D=0` (by `{FD.conj A D 0}`), and in turn `B=0` and `C=0` (by `{FD.disj B C D}`). Combination then adds the common constraint `C=0` back to the original space, without triggering further constraint propagation. Distribution on the remaining variables do not exhibit further information. Hence, saturation could neither prove nor disprove the formula unsatisfiable.

*Distribution.* The distribution strategy `SatDist` for 1-saturation as used in the above example takes as arguments a list of Boolean variables `Xs` to be distributed. For a single variable `X` three alternatives are needed. Two alternatives state that `X` takes either `0` or `1`. The third alternative allows to proceed without assigning a value to `X`. These three alternatives are programmed from two nested choices as sketched in Figure 7.

```
proc {SatDist Xs}
   ...
      choice choice X=0 [] X=1 end
      []     skip
      end
   ...
end
```

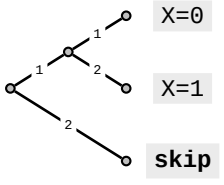



**Fig. 7.** Distribution for saturation.

To commit to the alternative for `X=0` (`X=1`), first the outer choice must be committed to its first clause, and then the inner choice must be committed to its first (second) clause. To proceed without assigning a value to `X`, the outer choice must be committed to its second clause. This protocol of committing choices will be implemented by the saturation engine.

Choice creation is iterated for all variables `X` in `Xs` not yet assigned a value until a fixed point is reached. The fixed point is reached when the number of variables not yet assigned a value has not changed after an entire iteration over all variables.

*Combination.* As distribution, combination is specific to the underlying constraint system. To parameterize the saturation engine with respect to combination, the engine takes a procedure for combination as input. The combination procedure for the Boolean case takes three lists `X0s`, `X1s`, and `Xs` of Boolean variables, of which `X0s` and `X1s` have been computed by the spaces obtained by distribution. `Xs` are the variables of the original space. A simple strategy is: If a variable has assigned the same value $n \in \{0,1\}$ in both `X0s` and `X1s`, the variable at that position in `Xs` is assigned to $n$.

```
fun {Sat S CB}
   case {Ask S}==alternatives then
      S0={Clone S} {Commit S0 1} {Commit S0 1}
      S1={Clone S} {Commit S1 1} {Commit S1 2}
   in
      case     {Ask S0}==failed then {Sat S1 CB}
      elsecase {Ask S1}==failed then {Sat S0 CB}
      else R0 R1 in
         {Merge S0 R0} {Merge S1 R1}
         {Inject S proc {$ R} {CB R0 R1 R} end}
         {Commit S 2} {Sat S CB}
      end
   else S
   end
end
```

**Fig. 8.** A generic saturation engine.

*The Saturation Engine.* The procedure `Sat` (shown in Figure 8) takes as input a space `S` and a procedure `CB` for combination. `Sat` returns a saturated computation space. If `S` is not distributable, it is returned. If `S` is distributable, two clones `S0` and `S1` of `S` are created and are committed to their appropriate alternatives (with the distribution strategy presented above, in `S0` (`S1`) the value `0` (`1`) is assigned to the variable `X`). If `S0` (`S1`) fails, saturation continues with `S1` (`S0`). Otherwise, combination is done as follows. Merging the spaces `S0` and `S1` makes their root variables with the corresponding constraints accessible by `R0` and `R1`. Execution of `CB` within `S` adds basic constraints on variables accessible from `R` and might trigger further constraint propagation in `S`.

The saturation engine is parametrized by the problem and how combination is done. Similar to search, the choices created by the distribution strategy comprise the interface between problem and saturation engine. The saturation engine as presented above can also be applied to non Boolean finite domain problems. The distribution strategy can still follow the same structure as presented above. However, it must employ constraints that fit the context of finite domain variables as alternatives. Combination in this context must also be generalized. Combination of the basic constraints $x \in D_0$ and $x \in D_1$ leads to the basic constraint $x \in (D_0 \cup D_1)$.

The engine only supports 1-saturation. To implement $n$-saturation for $n > 1$, the distribution strategy must be extended to recursively create additional choices. This also requires the saturation engine to handle these additional choices.

Since search engines as well as the saturation engine compute with spaces, it is straightforward to combine them. One possible approach is to first use saturation to infer as many additional basic constraints as possible, and then use search to actually solve the problem. A different approach is to interleave search and saturation.

## 12 Conclusion

We have presented first-class computation spaces as an abstraction to develop and program constraint inference engines at a high level. Computation spaces have been demonstrated to cover single, all, and best solution search, which are the inference engines found in existing constraint programming systems. Using computation spaces, we have developed inference engines for limited discrepancy search, visual search, and saturation. This has demonstrated that computation spaces can be applied to inference engines that go beyond existing constraint programming systems.

We are confident that computation spaces as high-level abstractions make it possible to adapt (like saturation) and invent (like visual search) further constraint inference methods. We expect that by this, computation spaces can contribute to the power of constraint programming.

## Acknowledgements

I am grateful to Martin Henz and Joachim Walser for frequent and fruitful discussions, especially on saturation. Martin Henz had the initial idea to use computation spaces for saturation. The search engine for LDS has been developed together with Joachim Walser. I would like to thank Martin Henz, Tobias Müller, Ralf Treinen, Joachim Walser, Jörg Würtz, and the anonymous referees for providing comments on this paper.

The research reported in this paper has been supported by the Bundesminister für Bildung, Wissenschaft, Forschung und Technologie (FKZ ITW 9601) and the Esprit Working Group CCL-II (EP 22457).

## References

1. Abderrahamane Aggoun, David Chan, Pierre Dufresne, Eamon Falvey, Hugh Grant, Alexander Herold, Geoffrey Macartney, Micha Meier, David Miller, Shyam Mudambi, Bruno

Perez, Emmanuel Van Rossum, Joachim Schimpf, Periklis Andreas Tsahageas, and Dominique Henry de Villeneuve. ECL$^i$PS$^e$ 3.5. User manual, European Computer Industry Research Centre (ECRC), Munich, Germany, December 1995.
2. Philippe Codognet and Daniel Diaz. Compiling constraints in `clp(FD)`. *The Journal of Logic Programming*, 27(3):185–226, June 1996.
3. James M. Crawford. An approach to resource constrained project scheduling. In George F. Luger, editor, *Proceedings of the 1995 Artificial Intelligence and Manufacturing Research Planning Workshop*, Albuquerque, NM, USA, 1996. The AAAI Press.
4. Mehmet Dincbas, Pascal Van Hentenryck, Helmut Simonis, Abderrahamane Aggoun, Thomas Graf, and F. Berthier. The constraint logic programming language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems FGCS-88*, pages 693–702, Tokyo, Japan, December 1988.
5. John Harrison. Stålmarck's algorithm as a HOL derived rule. In Joakim von Wright, Jim Grundy, and John Harrison, editors, *Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLs'96*, volume 1125 of *Lecture Notes in Computer Science*, pages 221–234, Turku, Finland, August 1996. Springer-Verlag.
6. William D. Harvey and Matthew L. Ginsberg. Limited discrepancy search. In Chris S. Mellish, editor, *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 607–615, Montreal, Canada, August 1995. Morgan Kaufmann Publishers.
7. Martin Henz, Martin Müller, Christian Schulte, and Jörg Würtz. The Oz standard modules. DFKI Oz documentation series, German Research Center for Artificial Intelligence (DFKI), Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany, 1997.
8. Martin Henz, Gert Smolka, and Jörg Würtz. Object-oriented concurrent constraint programming in Oz. In Vijay Saraswat and Pascal Van Hentenryck, editors, *Principles and Practice of Constraint Programming*, pages 29–48. The MIT Press, Cambridge, MA, USA, 1995.
9. ILOG. ILOG Solver: User manual, July 1996. Version 3.2.
10. Richard E. Korf. Depth-first iterative deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.
11. Christian Schulte. Oz Explorer: A visual constraint programming tool. In Lee Naish, editor, *Proceedings of the Fourteenth International Conference on Logic Programming*, pages 286–300, Leuven, Belgium, July 1997. The MIT Press.
12. Christian Schulte and Gert Smolka. Encapsulated search in higher-order concurrent constraint programming. In Maurice Bruynooghe, editor, *Logic Programming: Proceedings of the 1994 International Symposium*, pages 505–520, Ithaca, NY, USA, November 1994. The MIT Press.
13. Gert Smolka. The Oz programming model. In Jan van Leeuwen, editor, *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science*, pages 324–343. Springer-Verlag, Berlin, 1995.
14. Gunnar Stålmarck. A system for determining propositional logic theorems by applying values and rules to triplets that are generated from a formula. U.S. patent 5 276 897, 1994. Also as Swedish patent 467 076, 1991.
15. Joachim Paul Walser. Feasible cellular frequency assignment using constraint programming abstractions. In *Proceedings of the Workshop on Constraint Programming Applications, in conjunction with the Second International Conference on Principles and Practice of Constraint Programming (CP96)*, Cambridge, MA, USA, August 1996.
16. Jörg Würtz. Oz Scheduler: A workbench for scheduling problems. In *Proceedings of the 8th IEEE International Conference on Tools with Artificial Intelligence*, pages 132–139, Toulouse, France, November 1996. IEEE Computer Society Press.